

# **EXHIBIT C**

```
0 /* *****
1   File: sslrec.c
2
3   SSL Plus: Security Integration Suite(tm)
4   Version 1.1.1 -- August 11, 1997
5
6   Copyright (c)1996, 1997 by Consensus Development Corporation
7       Copyright (c)1997, 1998 by Aventail Corporation
8
9   Portions of this software are based on SSLRef(tm) 3.0, which is
10  Copyright (c)1996 by Netscape Communications Corporation. SSLRef(tm)
11  was developed by Netscape Communications Corporation and Consensus
12  Development Corporation.
13
14  In order to obtain this software, your company must have signed
15  either a PRODUCT EVALUATION LICENSE (a copy of which is included in
16  the file "LICENSE.TXT"), or a PRODUCT DEVELOPMENT LICENSE. These
17  licenses have different limitations regarding how you are allowed to
18  use the software. Before retrieving (or using) this software, you
19  *must* ascertain which of these licenses your company currently
20  holds. Then, by retrieving (or using) this software you agree to
21  abide by the particular terms of that license. If you do not agree
22  to abide by the particular terms of that license, then you must
23  immediately delete this software. If your company does not have a
24  signed license of either kind, then you must either contact
25  Consensus Development and execute a valid license before retrieving
26  (or using) this software, or immediately delete this software.
27
28  *****
29
30  File: sslrec.c      Encryption, decryption and MACing of data
31
32  All the transformations which occur between plaintext and the
33  secured, authenticated data that goes out over the wire. Also,
34  detects incoming SSL 2 hello messages and hands them off to the SSL 2
35  record layer (and hands all SSL 2 reading & writing off to the SSL 2
36  layer)..
37
38  ***** */
39
40 /* #define HYPER_DEBUG 1 */
41
42 #ifdef HYPER_DEBUG
43 #include <stdio.h>
44 #endif
45
46 #ifndef _SSL_H_
47 #include "ssl.h"
48 #endif
49
50 #ifndef _SSLREC_H_
51 #include "sslrec.h"
52 #endif
53
54 #ifndef _SSLALLOC_H_
55 #include "sslalloc.h"
56 #endif
57
58 #ifndef _CRYPTYPE_H_
59 #include "cryptype.h"
60 #endif
61
62 #ifndef _SSLCTX_H_
63 #include "sslctx.h"
64 #endif
65
66 #ifndef _SSLALERT_H_
67 #include "sslalert.h"
68 #endif
69
70 #ifndef _SSL2_H_
```

```
71 #include "ssl2.h"
72 #endif
73
74 #include <string.h>
75
76 static SSLerr DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx);
77 static SSLerr VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext
    *ctx);
78 static SSLerr ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer
    secret, CipherContext *cipherCtx, SSLContext *ctx);
79 static uint8* SSLEncodeUInt64(uint8 *p, uint64 value);
80
81 /* ReadSSLRecord
82  * Attempt to read & decrypt an SSL record.
83  */
84 SSLerr
85 SSLReadRecord(SSLRecord *rec, SSLContext *ctx)
86 {
87     SSLerr      err;
88     uint32      len, contentLen;
89     uint8       *progress;
90     SSLBuffer    readData, cipherFragment;
91
92     #ifdef HYPER_DEBUG
93     fprintf(stderr, "Got into SSLReadRecord, whee!\n");
94     #endif
95
96     /* if we get UDP data when we aren't expecting it, that's really bad,
97      so report an appropriate error. */
98     if((rec->contentType == SSL_application_data_ssloppy) &&
99         (! ctx->ssloppy))
100         return SSLProtocolErr;
101
102
103     if (!ctx->partialReadBuffer.data || ctx->partialReadBuffer.length < 5)
104     {
105         if (ctx->partialReadBuffer.data)
106             if ((err = SSLFreeBuffer(&ctx->partialReadBuffer, &ctx->sysCtx)) != 0)
107             {
108                 SSLFatalSessionAlert(alert_close_notify, ctx);
109                 return ERR(err);
110             }
111         if ((err = SSLAllocBuffer(&ctx->partialReadBuffer, DEFAULT_BUFFER_SIZE, &ctx->sysCtx))
112             != 0)
113         {
114             SSLFatalSessionAlert(alert_close_notify, ctx);
115             return ERR(err);
116         }
117     }
118
119     if (ctx->protocolVersion == SSL_Version_Undetermined ||
120         ctx->protocolVersion == SSL_Version_3_0_With_2_0_Hello)
121     {
122         if (ctx->amountRead < 1)
123         {
124             readData.length = 1 - ctx->amountRead;
125             readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
126             len = readData.length;
127             if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
128             {
129                 if (err == SSLWouldBlockErr)
130                     ctx->amountRead += len;
131                 else
132                     SSLFatalSessionAlert(alert_close_notify, ctx);
133                 return err;
134             }
135             ctx->amountRead += len;
136         }
137     }
138
139     /* In undetermined cases, if the first byte isn't in the range of SSL 3.0
140      * record types, this is an SSL 2.0 record
141      */
142     switch (ctx->protocolVersion)
143     {
144         case SSL_Version_Undetermined:
145         case SSL_Version_3_0_With_2_0_Hello:
146             if (ctx->partialReadBuffer.data[0] < SSL_smallest_3_0_type ||
147                 ctx->partialReadBuffer.data[0] > SSL_largest_3_0_type)
```

```
139         return SSL2ReadRecord(rec, ctx);
140     else
141         break;
142     case SSL_Version_2_0:
143         return SSL2ReadRecord(rec, ctx);
144     default:
145         break;
146 }
147
148
149 #ifdef HYPER_DEBUG
150     fprintf(stderr, "About to get into the read callback stuff\n");
151 #endif
152     if (ctx->amountRead < 5)
153     {   readData.length = 5 - ctx->amountRead;
154         readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
155         len = readData.length;
156         if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
157         {   if (err == SSLWouldBlockErr)
158             ctx->amountRead += len;
159         }
160         else if (err == SSLIOClosedOverrideGoodbyeKiss && ctx->amountRead ==
161             0)
162         {   SSLClose(ctx);
163             return SSLConnectionClosedGraceful;
164         }
165         else
166             return err;
167         ctx->amountRead += len;
168     }
169
170     ASSERT(ctx->amountRead >= 5);
171
172     progress = ctx->partialReadBuffer.data;
173     rec->contentType = *progress++;
174     if (rec->contentType < SSL_smallest_3_0_type ||
175         rec->contentType > SSL_largest_3_0_type)
176         return ERR(SSLProtocolErr);
177
178     rec->protocolVersion = (SSLProtocolVersion)SSLDecodeInt(progress, 2);
179     progress += 2;
180     contentLen = SSLDecodeInt(progress, 2);
181     progress += 2;
182     if (contentLen > (16384 + 2048)) /* Maximum legal length of an SSLCipherText payload */
183     {   SSLFatalSessionAlert(alert_unexpected_message, ctx);
184         return ERR(SSLProtocolErr);
185     }
186
187     if (ctx->partialReadBuffer.length < 5 + contentLen)
188     {   if ((err = SSLReallocBuffer(&ctx->partialReadBuffer, 5 + contentLen, &ctx->sysCtx)) !=
189         0)
190     {   SSLFatalSessionAlert(alert_close_notify, ctx);
191         return ERR(err);
192     }
193
194     if (ctx->amountRead < 5 + contentLen)
195     {   readData.length = 5 + contentLen - ctx->amountRead;
196         readData.data = ctx->partialReadBuffer.data + ctx->amountRead;
197         len = readData.length;
198         if (ERR(err = ctx->ioCtx.read(readData, &len, ctx->ioCtx.ioRef)) != 0)
199         {   if (err == SSLWouldBlockErr)
200             ctx->amountRead += len;
201         }
202         else
203             SSLFatalSessionAlert(alert_close_notify, ctx);
204         return err;
205     }
206     ctx->amountRead += len;
207 }
```

```
208     ASSERT(ctx->amountRead >= 5 + contentLen);
209
210     cipherFragment.data = ctx->partialReadBuffer.data + 5;
211     cipherFragment.length = contentLen;
212
213     /* Decrypt the payload & check the MAC, modifying the length of the buffer to indicate the
214      * amount of plaintext data after adjusting for the block size and removing the MAC
215      * (this function generates its own alerts)
216      */
217     if ((err = DecryptSSLRecord(rec->contentType, &cipherFragment, ctx)) != 0)
218         return err;
219
220     /* We appear to have successfully received a record; increment the sequence number */
221     if (rec->contentType != SSL_application_data_ssloppy)
222         IncrementUInt64(&ctx->readCipher.sequenceNum);
223
224
225 #ifdef SSL_COMPRESSION
226     if ((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
227         (ctx->selectedCompression->identifier != 0)) {
228
229         /* Allocate a buffer to return the plaintext in and return it */
230         if ((err = SSLAllocBuffer(&rec->contents, DEFAULT_BUFFER_SIZE,
231             &ctx->sysCtx)) != SSLNoErr) {
232             SSLFatalSessionAlert(alert_close_notify, ctx);
233             return ERR(err);
234         }
235         if ((err = ctx->selectedCompression->process(cipherFragment,
236             &(rec->contents),
237
238             ctx->readCompressRef,
239
240             ctx)) != SSLNoErr) {
241             SSLFreeBuffer(&rec->contents, &ctx->sysCtx);
242             SSLFatalSessionAlert(alert_decompression_failure, ctx);
243             return ERR(err);
244         }
245         #ifdef HYPER_DEBUG
246         fprintf(stderr, "Decompression created output of %d from size %d\n",
247             cipherFragment.length,
248             rec->contents.length);
249         #endif
250         } else {
251             if ((err = SSLAllocBuffer(&rec->contents, cipherFragment.length,
252                 &ctx->sysCtx)) != 0)
253             {
254                 SSLFatalSessionAlert(alert_close_notify, ctx);
255                 return ERR(err);
256             }
257             memcpy(rec->contents.data, cipherFragment.data, (size_t)
258                 cipherFragment.length);
259         }
260     #else
261     memcpy(rec->contents.data, cipherFragment.data, (size_t) cipherFragment.length);
262     #endif
263
264     ctx->amountRead = 0;          /* We've used all the data in the cache */
265     return SSLNoErr;
266 }
267
268 /* SSLWriteRecord does not send alerts on failure, out of the assumption/fear
269 * that this might result in a loop (since sending an alert causes SSLWriteRecord
270 * to be called).
271 */
```

```
269 SSLerr
270 SSLWriteRecord(SSLRecord rec, SSLContext *ctx)
271 {    SSLerr    err;
272     int        padding = 0, i, freerec = 0;
273     WaitingRecord *out, *queue;
274     SSLBuffer    buf, payload, secret, mac, nonce;
275     uint8        *progress;
276     uint16       payloadSize, blockSize, nonceSize = 0;
277
278     if (rec.protocolVersion == SSL_Version_2_0)
279         return SSL2WriteRecord(rec, ctx);
280
281     ASSERT(rec.protocolVersion == SSL_Version_3_0);
282     ASSERT(rec.contents.length <= 16384);
283
284 #ifdef SSL_COMPRESSION
285     if((ctx->compressNow) && (ctx->selectedCompression != NULL) &&
286         (ctx->selectedCompression->identifier != 0)) {
287         SSLBuffer compdata;
288
289         /* make a guess about how long the buffer will need to be */
290         if((err = SSLAllocBuffer(&compdata, rec.contents.length + 4,
291             &ctx->sysCtx)) != SSLNoErr)
292             return ERR(err);
293         if((err = ctx->selectedCompression->process(rec.contents, &compdata,
294
295             ctx->writeCompressRef,
296
297             ctx)) != SSLNoErr) {
298
299                 SSLFreeBuffer(&compdata, &ctx->sysCtx);
300                 return ERR(err);
301             }
302
303             rec.contents = compdata;
304             freerec = 1;
305         }
306 #endif
307
308     out = 0;
309     /* Allocate a WaitingRecord to store our ready-to-send record in */
310     if ((err = SSLAllocBuffer(&buf, sizeof(WaitingRecord), &ctx->sysCtx)) != 0)
311         return ERR(err);
312     out = (WaitingRecord*)buf.data;
313     out->next = 0;
314     out->sent = 0;
315
316     /* Allocate enough room for the transmitted record, which will be:
317     * 5 bytes of header +
318     * encrypted contents +
319     * macLength +
320     * padding [block ciphers only] +
321     * padding length field (1 byte) [block ciphers only]
322     */
323     payloadSize = (uint16) (rec.contents.length + ctx->writeCipher.hash->digestSize);
324     blockSize = ctx->writeCipher.symCipher->blockSize;
325     if (blockSize > 0)
326     {
327         padding = blockSize - (payloadSize % blockSize) - 1;
328         payloadSize = (uint16) (payloadSize + padding + 1);
329     }
330
331     if(ctx->ssloppy)
332     {
333         /* in this case we need more room, for the nonce */
334         nonceSize = (uint16) MAX(sizeof(uint64), ctx->writeCipher.symCipher->ivSize);
335         payloadSize += nonceSize; decided this was wrong logic */
336     }
337
338     out->data.data = 0;
```

```
334     if ((err = SSLAllocBuffer(&out->data, 5 + payloadSize + nonceSize,
335                               &ctx->sysCtx)) != 0)
336         goto fail;
337
338     progress = out->data.data;
339     *(progress++) = rec.contentType;
340     progress = SSLEncodeInt(progress, rec.protocolVersion, 2);
341     progress = SSLEncodeInt(progress, payloadSize, 2);
342
343     /* Copy the contents into the output buffer */
344     memcpy(progress, rec.contents.data, (size_t) rec.contents.length);
345     payload.data = progress;
346     payload.length = rec.contents.length;
347
348     progress += rec.contents.length;
349     /* MAC immediately follows data */
350     mac.data = progress;
351     mac.length = ctx->writeCipher.hash->digestSize;
352     progress += mac.length;
353
354     if (ctx->ssloppy)
355     {
356         uint64 noncevalue;
357
358         if ((err = SSLAllocBuffer(&nonce, nonceSize, &ctx->sysCtx)) != SSLNoErr)
359             goto fail;
360         if ((err = ctx->sysCtx.random(nonce, ctx->sysCtx.randomRef)) != SSLNoErr)
361             goto fail;
362
363         memcpy(&noncevalue, nonce.data, sizeof(noncevalue));
364
365         /* MAC the data, sloppy-style */
366         if (mac.length > 0) /* Optimize away null case */
367         {
368             secret.data = ctx->writeCipher.macSecret;
369             secret.length = ctx->writeCipher.hash->digestSize;
370             if ((err = ComputeMAC(rec.contentType, payload, mac, noncevalue,
371                                 secret, &ctx->writeCipher, ctx)) != 0)
372                 goto fail;
373         }
374
375         memcpy(progress, nonce.data, nonce.length);
376         progress += nonce.length;
377     }
378     else
379     {
380         /* MAC the data, normal mode */
381         if (mac.length > 0) /* Optimize away null case */
382         {
383             secret.data = ctx->writeCipher.macSecret;
384             secret.length = ctx->writeCipher.hash->digestSize;
385             if ((err = ComputeMAC(rec.contentType, payload, mac,
386                                 ctx->writeCipher.sequenceNum, secret,
387                                 &ctx->writeCipher, ctx)) != 0)
388                 goto fail;
389         }
390     }
391 }
392
393 /* Update payload to reflect encrypted data: contents, mac & padding */
394 payload.length = payloadSize;
395
396 /* Fill in the padding bytes & padding length field with the padding value; the
397  * protocol only requires the last byte,
398  * but filling them all in avoids leaking data
399  */
400 if (ctx->writeCipher.symCipher->blockSize > 0)
401     for (i = 1; i <= padding + 1; ++i)
402         payload.data[payload.length - i] = (uint8)padding;
403
404 /* Encrypt the data */
```

```
405     DUMP_BUFFER_NAME("cleartext data", payload);
406     if ((err = ctx->writeCipher.symCipher->encrypt(payload, payload,
407
                                                    ctx-
408
                                                    ctx-
409
                                                    ctx))
        != 0)
410         goto fail;
411
412     DUMP_BUFFER_NAME("encrypted data", payload);
413
414     /* Enqueue the record to be written from the idle loop */
415     if (ctx->recordWriteQueue == 0)
416         ctx->recordWriteQueue = out;
417     else
418     {
419         queue = ctx->recordWriteQueue;
420         while (queue->next != 0)
421             queue = queue->next;
422         queue->next = out;
423     }
424     if (ctx->ssloppy)
425         SSLFreeBuffer(&nonce, &ctx->sysCtx);
426     else
427         /* Increment the sequence number */
428         IncrementUInt64(&ctx->writeCipher.sequenceNum);
429
430     if (freerec)
431         SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
432
433     return SSLNoErr;
434
435 fail: /* Only for if we fail between when the WaitingRecord is allocated and when it is
        queued */
436     SSLFreeBuffer(&out->data, &ctx->sysCtx);
437     buf.data = (uint8*)out;
438     buf.length = sizeof(WaitingRecord);
439     SSLFreeBuffer(&buf, &ctx->sysCtx);
440     if (freerec)
441         SSLFreeBuffer(&(rec.contents), &ctx->sysCtx);
442     return ERR(err);
443 }
444
445 static SSLerr
446 DecryptSSLRecord(uint8 type, SSLBuffer *payload, SSLContext *ctx)
447 {
448     SSLerr err;
449     SSLBuffer content, nonce;
450
451     if (type == SSL_application_data_ssloppy)
452     {
453         nonce.length = MAX(sizeof(uint64), ctx->readCipher.symCipher->ivSize);
454         nonce.data = payload->data + (payload->length - nonce.length);
455         payload->length -= nonce.length;
456     }
457     if ((ctx->readCipher.symCipher->blockSize > 0) &&
458         ((payload->length % ctx->readCipher.symCipher->blockSize) != 0))
459     {
460         SSLFatalSessionAlert(alert_unexpected_message, ctx);
461         return ERR(SSLProtocolErr);
462     }
463     /* Decrypt in place */
464     DUMP_BUFFER_NAME("encrypted data", (*payload));
465
466     if (type == SSL_application_data_ssloppy)
467     {
468         if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, &nonce, ctx-
469             >readCipher.symCipherState, ctx)) != 0)
469         {
```

```
470             SSLFatalSessionAlert(alert_close_notify, ctx);
471             return ERR(err);
472         }
473     }
474     else
475     {
476         if ((err = ctx->readCipher.symCipher->decrypt(*payload, *payload, NULL, ctx-
477 >readCipher.symCipherState, ctx)) != 0)
478         {
479             SSLFatalSessionAlert(alert_close_notify, ctx);
480             return ERR(err);
481         }
482     }
483     DUMP_BUFFER_NAME("decrypted data", (*payload));
484
485     /* Locate content within decrypted payload */
486     content.data = payload->data;
487     content.length = payload->length - ctx->readCipher.hash->digestSize;
488     if (ctx->readCipher.symCipher->blockSize > 0)
489     {
490         /* padding can't be equal to or more than a block */
491         if (payload->data[payload->length - 1] >= ctx->readCipher.symCipher->blockSize)
492         {
493             SSLFatalSessionAlert(alert_unexpected_message, ctx);
494             return ERR(SSLProtocolError);
495         }
496         content.length -= 1 + payload->data[payload->length - 1]; /* Remove block size
497 padding */
498     }
499
500     /* Verify MAC on payload */
501     if (ctx->readCipher.hash->digestSize > 0) /* Optimize away MAC for null case */
502     {
503         if (type == SSL_application_data_ssloppy)
504         {
505             uint64 nonceNumber;
506
507             memcpy(&nonceNumber, nonce.data, sizeof(nonceNumber));
508             if ((err = VerifyMAC(type, content, payload->data + content.length,
509 nonceNumber, ctx)) != 0)
510             {
511                 SSLFatalSessionAlert(alert_bad_record_mac, ctx);
512                 return ERR(err);
513             }
514         }
515         else
516         {
517             if ((err = VerifyMAC(type, content, payload->data + content.length,
518 ctx->readCipher.sequenceNum, ctx)) !=
519 0)
520             {
521                 SSLFatalSessionAlert(alert_bad_record_mac, ctx);
522                 return ERR(err);
523             }
524         }
525     }
526
527     *payload = content; /* Modify payload buffer to indicate content length */
528
529     return SSLNoErr;
530 }
531
532 static uint8*
533 SSLEncodeUInt64(uint8 *p, uint64 value)
534 {
535     p = SSLEncodeInt(p, value.high, 4);
536     return SSLEncodeInt(p, value.low, 4);
537 }
538
539 static SSLerr
540 VerifyMAC(uint8 type, SSLBuffer data, uint8 *compareMAC, uint64 seqNo, SSLContext *ctx)
541 {
542     SSLerr err;
543     uint8 macData[MAX_DIGEST_SIZE];
544     SSLBuffer secret, mac;
545
546     secret.data = ctx->readCipher.macSecret;
```

```
538     secret.length = ctx->readCipher.hash->digestSize;
539     mac.data = macData;
540     mac.length = ctx->readCipher.hash->digestSize;
541
542     if ((err = ComputeMAC(type, data, mac, seqNo, secret,
543         &ctx->readCipher, ctx)) != 0)
544         return ERR(err);
545
546     if ((memcmp(mac.data, compareMAC, (size_t) mac.length)) != 0)
547         return ERR(SSLProtocolErr);
548
549     return SSLNoErr;
550 }
551
552 static SSLerr
553 ComputeMAC(uint8 type, SSLBuffer data, SSLBuffer mac, uint64 seqNo, SSLBuffer secret,
554     CipherContext *cipherCtx, SSLContext *ctx)
555 {
556     SSLerr err;
557     uint8 innerDigestData[MAX_DIGEST_SIZE];
558     uint8 scratchData[11], *progress;
559     SSLBuffer digest, scratch;
560
561 #ifdef HYPER_DEBUG
562     int i;
563     fprintf(stderr, "Buffer: ");
564     for(i = 0; i < data.length; i++)
565         fprintf(stderr, "%02x ", data.data[i]);
566     fprintf(stderr, "\n");
567     fprintf(stderr, "sequenceno: ");
568     for(i = 0; i < sizeof(uint64); i++)
569         fprintf(stderr, "%02x ", (unsigned char) *((unsigned char *) &seqNo) + i);
570     fprintf(stderr, "\n");
571     fprintf(stderr, "Secret: ");
572     for(i = 0; i < secret.length; i++)
573         fprintf(stderr, "%02x ", secret.data[i]);
574     fprintf(stderr, "\n");
575 #endif
576
577     ASSERT(cipherCtx->hash->macPadSize <= MAX_MAC_PADDING);
578     ASSERT(cipherCtx->hash->digestSize <= MAX_DIGEST_SIZE);
579     ASSERT(SSLMACPad1[0] == 0x36 && SSLMACPad2[0] == 0x5C);
580
581     if(cipherCtx->digestCtx.data == NULL) {
582         if ((err = SSLAllocBuffer(&cipherCtx->digestCtx,
583             cipherCtx->hash->contextSize, &ctx->sysCtx))
584             != 0)
585             goto exit;
586         cipherCtx->hash->create(cipherCtx->digestCtx);
587     }
588
589     /* MAC = hash( MAC_write_secret + pad_2 + hash( MAC_write_secret + pad_1 + seq_num + type +
590     length + content' ) ) */
591     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
592         goto exit;
593     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0) /* MAC secret */
594         goto exit;
595     scratch.data = SSLMACPad1;
596     scratch.length = cipherCtx->hash->macPadSize;
597     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* pad1 */
598         goto exit;
599     progress = scratchData;
600     progress = SSLEncodeUInt64(progress, seqNo);
601     *progress++ = type;
602     progress = SSLEncodeInt(progress, data.length, 2);
603     scratch.data = scratchData;
604     scratch.length = 11;
605     ASSERT(progress == scratchData+11);
606     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* sequenceNo,
type & length */

```

```
607     goto exit;
608     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, data)) != 0) /* content */
609         goto exit;
610     digest.data = innerDigestData;
611     digest.length = cipherCtx->hash->digestSize;
612     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, digest)) != 0) /* figure inner
digest */
613         goto exit;
614
615     if ((err = cipherCtx->hash->init(cipherCtx->digestCtx)) != 0)
616         goto exit;
617     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, secret)) != 0) /* MAC secret */
618         goto exit;
619     scratch.data = SSLMACPad2;
620     scratch.length = cipherCtx->hash->macPadSize;
621     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, scratch)) != 0) /* pad2 */
622         goto exit;
623     if ((err = cipherCtx->hash->update(cipherCtx->digestCtx, digest)) != 0) /* inner digest
*/
624         goto exit;
625     if ((err = cipherCtx->hash->final(cipherCtx->digestCtx, mac)) != 0) /* figure the mac */
626         goto exit;
627
628     err = SSLNoErr; /* redundant, I know */
629
630 exit:
631     return ERR(err);
632 }
```